Shading and Illumination
OpenGL Shading

Without Shading

With Shading
Physics
Bidirectional Reflectance Distribution Function (BRDF)

\[ f_r(\omega_i, \omega) = \frac{dL(\omega)}{L(\omega_i) \cos \theta_i d\omega_i} = \frac{dL(\omega)}{L(\omega_i) (-\omega_i \cdot n) d\omega_i} \]

n: Surface normal
L: Radiance (“intensity along a ray”)
Light Events

Reflection

- Bidirectional Reflectance Distribution Function (BRDF)

Refraction

- Bidirectional transmittance Distribution Function (BTDF)

Scattering

- Scattering Function

Bidirectional Scattering Distribution Function (BSDF)
Rendering Equation

• Given any point $x$ in a large environment, let $L(x, \omega)$ be the radiance along a ray, we have:

$$f_r(\omega_i, \omega) = \frac{dL(\omega)}{L(\omega_i)(-\omega_i \cdot n) d\omega_i}$$

$$L(x, \omega) = L_e(x, \omega) + \int_{\Omega} f_r(x, \omega_i, \omega) L(x, \omega_i)(-\omega_i \cdot n) d\omega_i$$

Le: Emitted intensity (light source)

$\Omega$: A hemisphere over $x$
Light Transport

- The rendering equation describes a light transport problem:
  - Each point has an integral equation.
  - Light can be reflected multiple times, so those integral equations are highly related.
  - Solution: global illumination.
  - Methods: ray tracing, radiosity.
Example: Caustics
Example: Color Bleeding
OpenGL Illumination

• Unfortunately, OpenGL doesn’t support global illumination.

• Instead, it only considers local illumination: light can only be reflected once.

• Even for local illumination, OpenGL is not physically based. It only provides an approximation.

• To accurately get multiple reflections, you need to better rendering algorithms.

• You can approximate some global illumination effects using GPU programmable shaders.
OpenGL Illumination

• Light sources
• Surface material properties
• Surface normal
• Incoming (light) and outgoing (eye) directions
Point Light Source

```c
GLfloat light_position[] = {1.0, 1.0, 1.0};
gllightfv(GL_LIGHT0, GL_POSITION, light_position);
```
Directional Light Source

GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
Spot Light Source

GLfloat light_position[] = {1.0, 1.0, 1.0};
gllightfv(GL_LIGHT0, GL_POSITION, light_position);
GLfloat spot_position[] = {1.0, 1.0, 1.0};
gllightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_position);
GLfloat EXPONENT=10;
gllightfv(GL_LIGHT0, GL_SPOT_EXPONENT, EXPONENT);
GLfloat cutoff=30;
gllightfv(GL_LIGHT0, GL_SPOT_CUTOFF, cutoff);
OpenGL separates surface reflection into three components:

1. Ambient
2. Diffuse
3. Specular

Therefore, you should define three light intensities. Each of them contributes to a reflection component:

- Ambient
- Diffuse
- Specular
OpenGL Light Source

```c
// To specify the light position and intensity
GLfloat LightPosition[] = {100, 0.0, 100};
GLfloat LightAmbient[] = {0.2, 0.2, 0.2};
GLfloat LightDiffuse[] = {1.0, 1.0, 1.0};
GLfloat LightSpecular[] = {0.4, 0.4, 0.4};

glLightfv(GL_LIGHT1, GL_POSITION, LightPosition);
glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, LightSpecular);
```
OpenGL Light Source

- For a point light source, the light intensity drops when the light moves away from the source. So OpenGL multiplies the intensity with an attenuation factor:

\[
\text{attenuation} = \frac{1}{A_{\text{constant}} + dA_{\text{linear}} + d^2A_{\text{quadratic}}}
\]

```c
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```
## Default Values

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Default Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_AMBIENT</td>
<td>((0.0, 0.0, 0.0, 1.0))</td>
<td>ambient RGBA intensity of light</td>
</tr>
<tr>
<td>GL_DIFFUSE</td>
<td>((1.0, 1.0, 1.0, 1.0))</td>
<td>diffuse RGBA intensity of light</td>
</tr>
<tr>
<td>GL_SPECULAR</td>
<td>((1.0, 1.0, 1.0, 1.0))</td>
<td>specular RGBA intensity of light</td>
</tr>
<tr>
<td>GL_POSITION</td>
<td>((0.0, 0.0, 1.0, 0.0))</td>
<td>((x, y, z, w)) position of light</td>
</tr>
<tr>
<td>GL_SPOT_DIRECTION</td>
<td>((0.0, 0.0, -1.0))</td>
<td>((x, y, z)) direction of spotlight</td>
</tr>
<tr>
<td>GL_SPOT_EXPONENT</td>
<td>0.0</td>
<td>spotlight exponent</td>
</tr>
<tr>
<td>GL_SPOT_CUTOFF</td>
<td>180.0</td>
<td>spotlight cutoff angle</td>
</tr>
<tr>
<td>GL_CONSTANT_ATTENUATION</td>
<td>1.0</td>
<td>constant attenuation factor</td>
</tr>
<tr>
<td>GL_LINEAR_ATTENUATION</td>
<td>0.0</td>
<td>linear attenuation factor</td>
</tr>
<tr>
<td>GL_QUADRATIC_ATTENUATION</td>
<td>0.0</td>
<td>quadratic attenuation factor</td>
</tr>
</tbody>
</table>

- No ambient
- Unit diffuse
- Unit specular
- Directional light along z
- No spot light
- No attenuation
Multiple Light Sources

```c
//Specify the position and properties for GL_LIGHT0
glEnable(GL_LIGHT0);
//Specify the position and properties for GL_LIGHT1
glEnable(GL_LIGHT1);
//Create other light sources here.
glEnable(GL_LIGHTING);

//You can turn on and off lights like this:
glDisable(GL_LIGHT0);
glEnable(GL_LIGHT0);
glDisable(GL_LIGHTING);
glEnable(GL_LIGHTING);
```
OpenGL Light Source

• The light source is subject to the MODELVIEW matrix as well.
  • You can create the light source in the eye space, so it always attached to the camera.
  • You can create the light source after camera motion, so it is fixed in the world space.
  • You can even apply extra transformation/animation to light source.
OpenGL Material

- OpenGL separates surface reflection into three components:
  - Ambient
  - Diffuse
  - Specular

- Therefore, you should define three material properties. Each of them contributes to a reflection component:
  - Ambient
  - Diffuse
  - Specular
Ambient Reflection

\[ L_{\text{ambient}} = I_{\text{ambient}} \cdot K_{\text{ambient}} \]

- Ambient Radiance (Pixel Intensity)
- Ambient Light
- Ambient Surface Material Coefficient

• The object will have a uniform color.
• It looks similar to the result without lighting.
• The color is not directly assigned. It’s controlled by the light source and surface reflection coefficient.
Diffuse Reflection

• Diffuse reflection assumes that the light is equally reflected in every direction.

• In other words, if the light source and the object has fixed positions in the world space, the camera motion doesn’t affect the appearance of the object.

• The reflection only depends on the incoming direction. It is independent of the outgoing (viewing) direction.

• Real objects with only a diffuse reflection property are called: Lambertian Surfaces, for example, clay.
Diffuse Reflection

• Lambert’s law says that the outgoing radiance depends on the cosine of the incident angle.
• Because the irradiance (photons per unit surface area) becomes smaller (as in the rendering function before).
**Diffuse Reflection**

\[ L_{\text{diffuse}} = I_{\text{diffuse}} \cdot K_{\text{diffuse}} \cos \theta_i \]

- **Diffuse Radiance**
- **Diffuse Light**
- **Diffuse Surface Material Coefficient**

**Diagram:**
- Brighter:
- Darker:
  \[ \theta_i \]
An Example

$K_{\text{diffuse}}$ vs $K_{\text{ambient}}$
Specular Reflection

• Some materials, such as plastic and metal, can have shiny highlight spots.

• This is due to a glossy mirror-like reflection.

• Materials have different shininess/glossiness.
Specular Reflection

• For a mirror, the light is reflected into a single direction.

• Specular reflection assumes that the glossy surface is not even, so the light is reflected into a small cone.

• If your viewing direction is close to the mirror direction, you receive high radiance.
Specular Reflection

- Given \( L, N \) and \( V \), we first calculate the mirror reflection direction:

\[
R = 2(L \cdot N)N - L
\]

- Then we have:

\[
L_{\text{specular}} = I_{\text{specular}} \cdot K_{\text{specular}} \cos^n \phi
\]

\( \phi \) is the angle between \( V \) and \( R \).

\[
\cos \phi = V \cdot R
\]

\( n \) indicates surface shininess.
An Example

$K_{\text{specular}}$

Shininess $n$
Total Reflection

- Illumination from a single light source $i$:

$$L_{\text{ambient}} = I_{\text{ambient}} \cdot K_{\text{ambient}}$$

$$L_{\text{diffuse}} = I_{\text{diffuse}} \cdot K_{\text{diffuse}} \cos \theta_i$$

$$L_{\text{specular}} = I_{\text{specular}} \cdot K_{\text{specular}} \cos^n \phi$$

$$L^i_{\text{total}} = L^i_{\text{ambient}} + L^i_{\text{diffuse}} + L^i_{\text{specular}}$$

- Illumination from multiple light source:

$$L_{\text{total}} = L_{\text{global ambient}} + L^0_{\text{total}} + L^1_{\text{total}} + L^2_{\text{total}} + L$$
OpenGL Material

• OpenGL defines material properties like this:

```c
glMaterialfV("face", "property", "value")
```

• Face: GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK
• Property: GL_AMBIENT, GL_DIFFUSE, ...
• Value: A vector array
For Example

```c
GLfloat mat_amb_diff[] = {1.0, 0.5, 0.8, 10.0};
GLfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0};
GLfloat shininess[] = {5.0};
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, mat_amb_diff);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, shininess);
```

- Specular highlights are usually in the light source’s color.
- Ambient and diffuse can be defined together or separately.
- If you don’t mind dark shadow, just ignore ambient.
- Shininess is from 0 (dull) to 128 (shiny).
Light Model

```c
glLightModelfv(property, value);
```

- Enable two-sided lighting
  - Property=GL_LIGHT_MODEL_TWO_SIDE
  - If value=0, only GL_FRONT is rendered;
  - If value=1, both GL_FRONT and GL_BACK are rendered

- Global ambient color
  - Property=GL_LIGHT_MODEL_AMBIENT
  - Value is a color array

- And more...
Surface Normal

- Normals are used in calculating reflection. Normals are also used to decide the surface front and back side.

\[ \mathbf{N} = \frac{\mathbf{E}_{10} \times \mathbf{E}_{20}}{|\mathbf{E}_{10} \times \mathbf{E}_{20}|} \]

\[ \mathbf{E}_{10} = \mathbf{V}_1 - \mathbf{V}_0 \]

\[ \mathbf{E}_{20} = \mathbf{V}_2 - \mathbf{V}_0 \]

It's normal is normalized \((x, y, z)\)
Surface Normal

• We can define a triangle normal for the whole triangle. The result is a surface made of triangles.

\[
\text{for(int } i=0; i<t\text{\_number}; i++)
\{
    \text{glNormal3f}(\ldots);
    \text{glBegin(GL\_TRIANGLES)};
    \text{glVertex3f}(\ldots);
    \text{glVertex3f}(\ldots);
    \text{glVertex3f}(\ldots);
    \text{glEnd}();
\}
\]
Surface Normal

• But the mesh represents a smooth surface. We define a normal for each vertex, by taking the normalized average of its adjacent triangle normals.

```cpp
for(int i=0; i<t_number; i++)
{
    glBegin(GL_TRIANGLES);
    glNormal3f(...); //XN0
    glVertex3f(...); //X0
    glNormal3f(...); //XN1
    glVertex3f(...); //X1
    glNormal3f(...); //XN2
    glVertex3f(...); //X2
    glEnd();
}
```
Lighting and Shading

- Each pixel corresponds to a visible 3D point.
- We need to calculate the pixel intensity for that 3D point, based on the reflection model.

**Ray Tracing**

- Given a 2D Pixel Location
  - Inverse Projection
  - Selection Ray
  - Intersection Test
  - A 3D point on a primitive
  - Reflection Calculation

**The Graphics Pipeline**

- Given a 3D primitive
  - Projection
  - A 2D primitive
  - Rasterization
  - A number of pixels
Lighting and Shading

- OpenGL does reflection calculation only for each vertex (unless using GPU shaders).

**Given a 3D primitive**

1. Projection
2. A 2D primitive (with 3D info)
3. Rasterization
4. A number of pixels (with 3D info)
5. Fragment Shader
   - Pixels with intensities
   - **OpenGL + GPU**

**Given a 3D polygon**

1. Lighting
2. A 3D polygon, with intensity info
3. Projection
4. Interpolation/Rasterization
5. A 2D polygon, with intensity info
6. Pixels with intensities
   - **OpenGL**
OpenGL Shading

• Now given a 2D polygon with intensity information on each vertex, how to get the value for each pixel?
  • Flat shading
  • Gouraud Shading*
  • Phong Shading*

Given a 3D polygon

Lighting

A 3D polygon, with intensity info

Projection

A 2D polygon, with intensity info

Interpolation/Rasterization

Pixels with intensities

*: Both were proposed by researchers at the University of Utah in 1970s.
Flat Shading

- Flat shading uses one vertex information for the whole primitive.

```c
glShadeModel(GL_FLAT)
```

What’s the difference?

Vertex Normal + Flat Shading  Triangle Normal + Smooth Shading
Flat Shading VS. Smooth Shading

• Flat shading has this discontinuous artifact.
  • This is fast, since only needs one calculation per polygon.
  • Only useful when the polygons are very small.

• Smooth shading avoid the discontinuous artifact.
  • Slower
  • More realistic.
Smooth Shading

- OpenGL uses a smooth Gouraud shading technique.
- The basic idea is to calculate the color intensity at each vertex, and then interpolate over the polygon.
- To use smooth shading:

```plaintext
glShadeModel(GL_SMOOTH)
```
Review: Linear Interpolation

• Given a line segment with two end points (P and Q) and a sample point R, we can interpolate a value function V over the line as:

\[ V_r = \frac{|Q - R|V_p + |P - R|V_q}{|P - Q|} \]
OpenGL Interpolation

- OpenGL does 2D interpolation in a scanline way.
- The result is identical to the interpolation using barycentric coordinates. But the scanline method is faster. Why?
Gouraud Shading

- Gouraud shading calculates the color per vertex.
- It has artifacts too.
- To solve it, use smaller polygons.
Phong Shading

• Instead of interpolating the color (light intensity), Phong shading interpolates surface normals, and then calculates the 3D reflection per 2D pixel.
• OpenGL doesn’t support it, since 3D information needs to be carried through the whole pipeline.
• GPU supports it. You can use OpenGL shading language (GLSL).
• It’s still under the projection+scanline framework, but its result is comparable to ray tracing.
Summary

• The projection+scanline graphics pipeline is also called rasterization-based rendering.
  • Examples: OpenGL, Pixar Renderman, GPU...
  • Fast, but difficult to handle global illumination
  • Widely used in games

• Alternatively, ray tracing is based on the inverse-projection+ray-intersection framework
  • Examples: POV-Ray, new Pixar Renderman, PBRT...
  • Slow, but handles global illumination
  • Used for offline rendering, such as movie effects